Preprint
UCRL-ID-153725

Investigate Methods to Decrease Compilation Time-AX-Program Code Group

Teresa Cottom

Computer Science R & D Project

June 11, 2003

U.S. Department of Energy

Lawrence
Livermore
National
Laboratory

Investigate Methods To Decrease Compilation Time
AX-Program Code Group Computer Science R&D Project

Teresa L Cottom
cottomt@llnl.gov
11/20/03 - 04/03/03

Proposal:

Large simulation codes can take on the order of hours to compile from scratch. In Kull, which uses generic programming techniques, a significant portion of the time is spent generating and compiling template instantiations. I would like to investigate methods that would decrease the overall compilation time for large codes. These would be methods which could then be applied, hopefully, as standard practice to any large code. Success is measured by the overall decrease in wall clock time a developer spends waiting for an executable.

Background:

As a developer of Kull, I quickly found myself frustrated waiting for compilations of the code. On OSF platforms, the code would take 3 _ hours to build from scratch and 2 _ hours on AIX platforms. These times were using the maximum number of processors available via the -j# argument on both platforms. Motivation for decreasing these times is easily apparent, as it would increase not only my own productivity, but the productivity of all developers on the team. In the past, I had worked with build systems which used precompiled headers and used distributed compilations to increase efficiency.

Overview:

Having worked in the past with compilers that use precompiled headers to minimize the header file processing for recompiles, I explored the use of compiler options in KCC. I first investigated the use of precompiled headers to decrease compilation times. Although the precompiled headers reduced compilation times for recompiles, the first time compilation times increased and the precompiled headers required on the order of megabytes of disk space per header file. Kull is a heavily templated code which may be the cause / effect relation but since there was such a time and space overhead, I discarded thoughts of using precompiled headers.

Kull initially used a python script for generating object file dependency information for C and C++ source files. This required the python script to recursively parse through all included header files to track the dependency. The header files were of course then parsed a second time by the compiler for compilation of the source files. Searching through the compiler options, I was pleased to find compiler options for generating dependency information for all compilers that we use:

g++	-MD	<sourcefile>.d
pgCC	-MD	<sourcefile>.d
KCC	--output_dependencies <filename>	<filename>

By using these compiler options when the object file is generated, the dependency file is written during the same step as the object file. This made the need for the python script obsolete and now only required a single recursive processing of the header files. This improvement reduced the build times by an average of 5 seconds per source file. In Kull, this netted a savings of about 10 minutes.

Along the lines of dependency file generation, it was noted that a second python script was being used to track f90 module dependencies. This python script operated on an entire subdirectory, halting the compilation of any source files in the subdirectory until the script was completed. The python script was modified such that it only operated on a single source file, and also so that it tracked include files dependencies enabling correct object file recompilation. Modifying the script in this fashion allowed gmake to parallelize the generation of FORTRAN dependency files by spawning off multiple python scripts simultaneously. This reduced the wall clock time spent building dependency information, allowing gmake to more quickly get to spawning off compilations.

Further exploring compiler options in KCC, I uncovered the --parallel_build # option which uses # number of threads to parallelize the generation of template instantiations. Since template instantiations are done mostly at library and executable link times, it made sense to add --parallel_build 2 to the link flags. This gave some improvements to the speed of generating the library and executable files as now the compiler was using two threads instead of one to generate instantiations, occasionally taking advantage of an idle processor.

By timing the compilations when making these compiler flag modifications, it was noted on AIX that although gmake was invoked as gmake -j14, the time command informed me that only 415% of cpu utilization was used on average for the life of the build. This would be out of a possible 1400% since there are 16 processors and 14 were requested. It should also be noted that these compilations were invoked at an obscenely early hour in the morning while I was the only user running active commands. This terribly inadequate use of resources prompted me to analyze gmake in regards to the Kull make system.

In my analysis, I concluded that gmake will determine the number of targets which can currently be built and evenly distribute the available threads between these targets. The original Kull make system required a change of directory to the lowest subdirectory, and then would invoke make in that subdirectory and work its way back up. Each subdirectory would get all 14 threads one at a time since they were the only 'top' level target that required building. Some subdirectories had less than 14 source files, which would result in idle threads. Some subdirectories had a combination of large and small source files, which would result in threads idling waiting for the slower compiles to finish. All subdirectories would have a minimum number of link targets (<= 4) which would result in many idle threads. Since the link steps involve the template instantiations, this is by far the longest step, and idling 10 to 13 threads really hurt the performance of building Kull.

Knowing that gmake will distribute the threads evenly to the number of targets requiring building, I began restructuring the targets in the Kull make system. If the 'true' library and executable targets could be built in parallel, then since Kull has many more than 14 'true' targets, gmake would give each target a thread and would build the targets in parallel. As gmake would complete the smaller library targets, it would then invoke the building of another library target with the available thread. To maximize the parallelization of building the libraries, the system needs to ensure that the 'heavy-weight' targets begin building early on. Performing extra analysis at the configuration stage, 'heavy weight' targets were determined as those with 13 or more source files.

The Kull make system stored all targets in the same subdirectory within the same Makefile. This required duplication of object rules and flags for each target in the Makefile. The restructuring of the makefiles included separating out library and executable targets into individual makefiles that were based off of 'template' makefiles. Similarities in the makefiles were extracted out and placed in shared included makefiles. These included:

Make.DependRules	Contains rules to include/create dependencies
Make.CXXFlags	Contains C and C++ compiler flags
Make.CXXRules	Contains rules to create object files
Make.FortranFlags	Contains FORTRAN compiler flags
Make.FortranRules	Contains rules to create object files
Make.StaticLibraryRules	Contains rule to create library target

The individual makefiles would define its own subset of variables which would be used in these included makefiles.

```
NAME=
SOURCES=
EXTRACXXFLAGS=
INCLUDEDIRS=
```

Restructuring the makefiles in this way, reduced the size of the Makefiles in Kull from about 24k, to 4k. Although most of the 24k of the original Makefiles was duplication between the subdirectories, the Makefiles could not be cached because they were different files. The system could now cache these similarities as they are separated out and the same exact files are reused. These individual makefiles were also updated for an 'objs' target which would simply compile the source files in the directory and not build any linkable targets. This could be seen as a 'quick' make, setting aside the timely 'link' build for a later time.

The top level Makefile was updated for inclusion of another makefile which defined remote targets which were used to build sources or libraries of an individual target. Each target would have two remote targets like:

```
rfoo1objs:
        (cd src/foos; $(MAKE) objs -f Makefile.foo1)
rfoo1:
        (cd src/foos; $(MAKE) -f Makefile.foo1)
```

The configuration system was updated to determine 'heavy weight' targets. By stacking the targets such that only a single target is determined to be ready to build at a time, gmake would give all its threads to the single target for building. In this fashion, 'heavy weight' targets would build their source files in parallel, and then allow the next 'heavy weight' target to do the same. When all 'heavy weight' targets are completed, then gmake can begin building the 'true' targets in parallel. Termed 'rack and stack' the target organization went as follows:

```
quick : rlibs

rfoo1:
        (cd src/foos; $(MAKE) -f Makefile.foo1)
rfoo2:
        (cd src/foos; $(MAKE) -f Makefile.foo2)
rbar1:
        (cd src/bars; $(MAKE) -f Makefile.bar1)
rbar2:
        (cd src/bars; $(MAKE) -f Makefile.bar2)

rlibs = rfoo1 rfoo2 rbar1 rbar2

rfoo1objs:
        (cd src/foos; $(MAKE) objs -f Makefile.foo1)
rfoo2objs: rfoo1objs
        (cd src/foos; $(MAKE) objs -f Makefile.foo2)
robjs : rfoo2objs
rlibs : robjs
```

Using the 'quick' target, gmake would follow along the 'rack and stack'. At first, only the rfoo1objs target is dependent free and buildable, so gmake passes along all its threads and invokes the rfoo1objs make command. This 'heavy weight' target now has its source files compiled in parallel. Upon completion, now rfoo2objs is the only target dependent free and buildable. Again, gmake passes along all its threads to the rfoo2objs make command and those sources are built in parallel. Now, robjs is free of dependencies which in turn means rlibs is free of dependencies. There are now four targets free of dependencies and ready to build, rfoo1 rfoo2 rbar1 and rbar2. The number of threads are evenly distributed amongst these targets and they are invoked. These 'true' targets are now built in parallel. The 'heavy weight' targets rfoo1 and rfoo2 are really only completing the link steps, and since the foo targets are 'heavy weights' this is a longer process than for that of the bar targets. Reorganizing the makefiles in such a way resulted in enormous wall clock time savings due to maximizing the parallelization of the threads. The overall build now capitalized 950% cpu utilization for the life of the build on AIX. This gave the greatest increase in compilation time savings on AIX and OSF.

Many Kull developers build on AIX since there are 16 processors on the nodes, where as OSF only has a maximum of 4 processors per node. If the number of available processors for building on OSF could be increased, this could encourage more developers to work on OSF and decrease the load on AIX.   A way to increase this would be to allow builds across multiple nodes. If you could use 4 machines each with 3 processors, the capacity goes from 3 cpus to 12. An earlier attempt at this by another party was pmake. I determined not to use pmake as it was a specific older version of gmake source code with embedded MPI calls. Instead I preferred to leave the multinode build open to any version of gmake. This would be done by running gmake in --dry_run mode to determine which targets need rebuilding, its subdirectory, and the commands to build the target.

In using pyMPI in Kull itself, I became familiar with its MPI interface and enjoyed its simplicity. With pyMPI, I remove any need to the low-level MPI, and I depend fully upon a vendor library for various platform support. So, it was decided that a python script would be the best resolution for the multinode build driver.   The goal was also to leave this driver as a generic implementation that would simply parallelize 'tasks' to the 'workers' available. A task is simply a directory and a command, and the command is invoked from the given directory.   A 'worker' is any MPI process that is not rank 0. Rank 0 is reserved as the 'manager.'

The python gmake driver was implemented as a class which invokes gmake in dry run mode and builds a hash table of targets which stores the target dependency information, and commands. The class then exposes a function, getNextCommand() which determines the next dependency free target and returns its 'task.'

To start the multinode build, mpirun is started given the machine file list, the number of processors to be used, a path to pyMPI, the pyMPICommands.py python script and a list of targets to make. The 'manager' then creates a GmakeCommand instance which runs gmake in dry run mode. The 'workers' start up and request a 'task' from the 'manager.' The 'manager' gets tasks via the CommandManger.getNextCommand() and shovels these jobs off to the 'workers' until there are no more commands to complete.

Using this process, I introduced a new target in the Kull make system called multinode which invokes the pyMPI multinode build with the correct machine host file for the platform.   Now, the default OSF configuration makes 12 processors available for compilation via the multinode target. This gives significant speed improvement for building on OSF and will come in handy when Kull is ported to Linux where there are only two processors available per node.   This is used on AIX to load balance between nodes available to Kull developers with a default of 16 processors used.

The final compilation improvement was to build shared libraries on OSF using KCC as previously Kull only supported building static libraries.   Building shared libraries with KCC also required removing the --one_per_instantiation flag option which is used for static library builds.   This gave significant object file generation time improvements as now less templates were being instantiated per object file. Finally, by having shared libraries the Kull executable need only be linked once, not every time an individual library requires rebuilding.   In obsoleting the need to relink, savings of three to four minutes are acquired for every recompile.

Summary:

Analyzing the make system of a slow to build project can benefit all developers on the project. Taking the time to analyze the number of processors used over the life of the build and restructuring the system to maximize the parallelization can significantly reduce build times. Distributing the build across multiple machines with the same configuration can increase the number of available processors for building and can help evenly balance the load. Becoming familiar with compiler options can have its benefits as well. The time improvements of the sum can be significant.

Initial compilation time for Kull on OSF1 was 3 _ hours. Final time on OSF1 after completion is 16 minutes. Initial compilation time for Kull on AIX was 2 _ hours. Final time on AIX after completion is 25 minutes.

Developers now spend 3 hours less waiting for a Kull executable on OSF1, and 2 hours less on AIX platforms. In the eyes of many Kull code developers, the project was a huge success.

Finally, if you find that you spend too much time counting the pixels on your monitor, there is probably some analysis and improvements that can be made in your system as well.

Future Work:

The Gmake class used to determine compilation commands can easily be upgraded to be multi-threaded which would minimize the time needed to process gmake output before distributing compilation commands to processors.

References:

http://www.gnu.org/manual/make/html_mono/make.html
http://www.python.org/
http://www.sourceforge.net/projects/pympi

Thanks to the following for promptly answering various implementation specific MPI questions:

John Gyllenhaal
Linda Stewart

University of California
Lawrence Livermore National Laboratory
Technical Information Department
Livermore, CA 94551